On-the-Fly Controller Synthesis for Discrete and Dense-Time Systems

Stavros Tripakis and Karine Altisen

[1] Verimag, currently at UC Berkeley,
stavros@eecs.berkeley.edu
[2] Verimag,
altisen@imag.fr

Abstract. We present novel techniques for efficient controller synthesis for untimed and timed systems with respect to invariance and reachability properties. In the untimed case, we give algorithms for controller synthesis in the context of finite graphs with *controllable* and *uncontrollable* edges, distinguishing between the actions of the system and its environment, respectively. The algorithms are *on-the-fly*, since they return a controller as soon as one is found, which avoids the generation of the whole state space.
In the timed case, we use the model of *timed automata* extended with controllable and uncontrollable discrete transitions. Our controller-synthesis method here is only half on-the-fly, since it relies on the a-priori generation of a finite model (graph) of the timed automaton, as quotient of the *time-abstracting bisimulation*. The quotient graph is essentially an untimed graph, upon which we can apply the untimed on-the-fly algorithms to compute a timed controller.

Keywords. Controller Synthesis, On-The-Fly Algorithms, Timed Automata, Time-Abstracting Bisimulation.

1 Introduction

An embedded system can be usually considered as a reactive machine that cooperates with an environment to provide a service. The environment generates input events triggering actions that change the state of the machine, which may in turn produce output events that affect the environment. The formal analysis of such systems requires models and techniques that take into account not only the properties of the embedded machine, but also the characteristics of the environment, and in particular, the unpredictable (sometimes even hostile) nature of the latter.

In formal verification (or *model-checking*) there is no distinction between actions of the system and those of the environment. Usually the model of the system is “closed” by descriptions of the environment and the model-checking procedure checks that *all* execution sequences of the closed model satisfy a set of properties. In such an approach, the description of the system must be complete (often deterministic) at the time of verification, and there is little or no possibility at all of exploiting non-determinism of the system's description to perform "fine-tuning" of the application.

An alternative approach is to start from an "open" model where actions of the system and actions of the environment are distinguished. Such a model can be considered as "incomplete" in the sense that it describes a more liberal behavior and usually the question arises of "closing" the system so that the requirements are met. This is the problem of *controller synthesis* [RW87]. It consists in computing a controller which observes the state/actions of the environment and restricts the choices of the system, ensuring that the given property is satisfied no matter how the environment behaves.

Although more interesting (and more general) than verification, controller synthesis is also a more difficult problem. On the other hand, a number of heuristics which have improved the efficiency of model-checking, such as *on-the-fly* techniques, have not been developed, to our knowledge, in the context of controller synthesis. This is perhaps the reason why the latter has not found so much application in practice as model-checking has.

In this paper we propose on-the-fly methods for controller synthesis for discrete (i.e., finite-state) and dense-time systems, with respect to *invariance* and *reachability* properties. A controller for invariance tries to keep the system inside a set of "safe" states. A controller for reachability tries to lead the system to a set of "target" states.

For the description of discrete systems we use *game graphs* (GG), that is, finite graphs with edges marked as *controllable* or *uncontrollable*, modeling the actions of the system (and possible choices of the controller) and those of the environment, respectively. For game graphs we define the notion of *strategies*, which are sub-graphs representing the choices of the controller for each possible choice of the environment. Controller synthesis consists in computing a strategy with respect to invariance (all nodes are safe) or reachability (all paths lead to the target nodes).

Our method in the untimed case is fully on-the-fly, that is, game graphs can be implicitly represented using a higher-level formalism and generated at the same time as the calculation of the strategy. A strategy is returned as soon as it is found, which means that the whole state space does not necessarily have to be generated. The method is based on a forward reachability analysis using a depth-first search and its worst case complexity is $O(n + m)$, where n and m is, respectively, the number of nodes and edges in the graph.

In the timed case, we describe systems using the model of *game timed automata* (GTA) [MPS95], [AMP95], which are simply timed automata (TA) [ACD93], [HNSY94] with discrete transitions marked as controllable or uncontrollable. We define *timed strategies*, a notion similar to the untimed case but adapted to take into account the density of the time domain, as well as the fact that "time can be in favor of both the controller and the environment" [MPS95].

Our controller-synthesis method in the timed case is only half on-the-fly, since it involves two steps: first, we generate a finite-state model of the GTA (a graph); then we apply the on-the-fly untimed synthesis algorithms on this graph. The latter is the *quotient* of the GTA with respect to the *time-abstracting bisimulation* defined in [TY96]. This equivalence abstracts away exact time delays while preserving all properties of interest. We show how the time-abstracting quotient graph of a GTA can be viewed as a game graph so that GTA controller synthesis is reduced to GG controller synthesis.

We illustrate our approach on a toy-example, the Train-Gate-Controller system of [Alu91] (viewed as a GTA). We show how a strategy can be obtained for this system in an on-the-fly manner, that is, without having to explore the whole quotient graph.

Relation to the Literature

Controller synthesis is close to the theory of *games*. In the domain of formal methods, pioneering have been the works of [RW87, PR89], who studied the problem in the untimed case. Algorithms for this theory have been given based on a backward fix-point calculation of a predecessor operator which returns the set of states which can be led to a set of target states independently of uncontrollable actions. *Symbolic* versions of this algorithm (i.e., reasoning in terms of sets of states instead of single states) have been presented in [HW92, Le 93, MPS95, AMP95] and prototype implementations in [WM99]. The fix-point algorithms are not on-the-fly, since the fix-point calculation has to terminate in order for the (maximal) strategy to be computed. Moreover, the method is based on a predecessor operator, therefore, may needlessly consider states which are not reachable. To our knowledge, on-the-fly algorithms for controller synthesis have not been presented before in the literature.

In the timed case, [HW91] use the framework of [RW87] to solve controller synthesis for deterministic TA. [MPS95] present a fix-point controller-synthesis algorithm for general TA and with respect to a large class of properties, including invariance and reachability. The main drawback in the above works is the implementation of the symbolic predecessor operator, which is expensive in the dense-time case (it involves the exponential-cost operation of complementation of polyhedra).

Organization of the Paper

In section 2, we treat the problem in the untimed case. We introduce game graphs and strategies, we define the problem of controller synthesis in terms of a search for strategies and we present the two on-the-fly algorithms for strategies with respect to invariance and reachability. In section 3, we review timed automata and the time-abstracting bisimulation and define the quotient graph with respect to this relation. In section 4, we extended timed automata to game timed automata and define timed strategies and controller synthesis in the timed case. We also show how the on-the-fly algorithms of section 2 can be applied on the quotient graph in order to solve the problem for the timed case. Section 5 presents our conclusions.

2 On-the-Fly Controller Synthesis for Finite Discrete-State Systems

In this section we give an on-the-fly solution to the controller-synthesis problem for the untimed case. We first present our model and its semantics in terms of strategies. Then we give two algorithms for computing on-the-fly strategies with respect to invariance or reachability.

2.1 The Model: Finite Graphs with Controllable/Uncontrollable Edges

We would like to describe finite-state systems which involve an unpredictable (or even hostile) environment. To model such systems we use finite graphs the edges of which are labeled as controllable or uncontrollable, to model the actions of the system and its environment, respectively.

More formally, a *game graph* (GG) is a finite labeled graph $G = (V, v_0, \rightarrow)$, where V is a finite set of nodes, $v_0 \in V$ is the initial node and $\rightarrow \subseteq V \times \{c, u\} \times V$ is a set of edges. For two nodes $v, w \in V$, we write $v \xrightarrow{c} w$ (resp. $v \xrightarrow{u} w$) if $(v, c, w) \in \rightarrow$ (resp. $(v, u, w) \in \rightarrow$). An edge $v \xrightarrow{c} w$ is called *controllable* and w is a controllable *successor* of v. An edge $v \xrightarrow{u} w$ is called *uncontrollable* and w is a uncontrollable successor of v.

We assume that a GG does not contain any nodes without any controllable successor, that is, for each $v \in V$ there exists $w \in V$ such that $v \xrightarrow{c} w$. This is not a restriction to the model since "dummy" self-looping controllable edges can be added to nodes without controllable successors. Notice that this assumption implies that a GG does not contain any *sink* nodes, that is, nodes with no successors.

Example. Figure 1 shows two game graphs. Their nodes are numbered 0 to 3 and 0 to 4, respectively.

Remark 1. The model of game graphs is at least as general as the game-theoretic model where controllable and uncontrollable actions *alternate*, that is, AND-OR graphs: AND nodes correspond to game-graph nodes having only uncontrollable successors (except the self-loop controllable edge) and OR nodes correspond to game-graph nodes having only controllable successors.

Strategies. We are interested in controlling game graphs with respect to two types of properties, namely, *invariance* (where the controller tries to keep the system inside a set of safe states) and *reachability* (where the controller tries to lead the system to a set of target states).

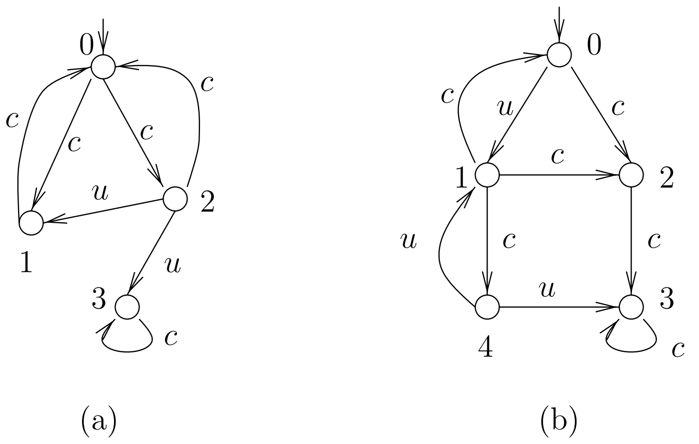

Fig. 1. Two game graphs.

Consider a GG $G = (V, v_0, \rightarrow)$ and a set of nodes $\hat{V} \subseteq V$.

A *strategy of* G *with respect to invariance of* $\hat{V}$ is a graph $G_1 = (V_1, v_0, \rightarrow_1)$ such that:

1. $V_1 \subseteq \hat{V}$ and $\rightarrow_1 \subseteq \rightarrow$.
2. For each node $v \in V_1$, if $v \xrightarrow{u} w$, then $w \in V_1$ and $v \xrightarrow{u}_1 w$.
3. For each node $v \in V_1$, there is an edge $v \xrightarrow{c}_1 w$.

In other words, a strategy with respect to invariance is a subgraph of G restricted to the "safe nodes" $\hat{V}$ (condition 1), and such that for each node, all its uncontrollable successors (condition 2) and at least one of its controllable successors (condition 3) are kept. Condition 1 also ensures that the system remains in the set of safe states. All nodes in V_1 are said to be *winning* with respect to invariance of $\hat{V}$.

A *strategy of* G *with respect to reachability of* $\hat{V}$ is a graph $G_1 = (V_1, v_0, \rightarrow_1)$ such that:

1. $V_1 \subseteq V$ and $\rightarrow_1 \subseteq \rightarrow$.
2. For each node $v \in V_1 \setminus \hat{V}$, if $v \xrightarrow{u} w$, then $w \in V_1$ and $v \xrightarrow{u}_1 w$.
3. For each node $v \in V_1 \setminus \hat{V}$, there is an edge $v \xrightarrow{c}_1 w$.
4. For each node $v \in V_1$, there exists a path $v \xrightarrow{c}_1 v_1 \xrightarrow{c}_1 \cdots \xrightarrow{c}_1 v_n$ such that $v_n \in \hat{V}$.

In other words, a strategy with respect to reachability is a subgraph of G (condition 1) such that each node can reach the "target nodes" $\hat{V}$ by a path of controllable edges (condition 4), and for each non-target node, all its uncontrollable successors (condition 2) and at least one of its controllable successors (condition 3) are kept. Condition 4 ensures that the controller can lead the system to the set of target states. All nodes in V_1 are said to be *winning* with respect to reachability of $\hat{V}$.

Example. Figure 2(a) shows a strategy for the game graph of figure 1(a) with respect to invariance of $\{0, 1, 2\}$. Notice that the controllable edge $0 \xrightarrow{c} 2$ is eliminated, since from node 2 the system can be led to the unwanted node 3 by an uncontrollable edge.

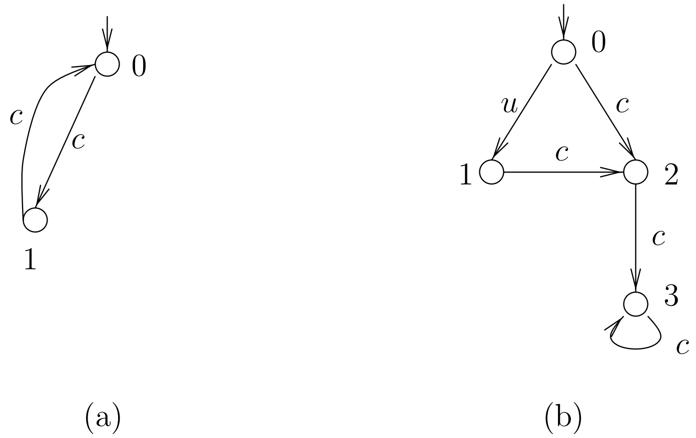

Fig. 2. Two strategies.

Figure 2(b) shows a strategy for the game graph of figure 1(b) with respect to reachability of $\{3\}$. The controller, being at node 1, chooses to move to node 2, from where reachability of node 3 is guaranteed.

Controller Synthesis for Game Graphs. Given a GG $G = (V, v_0, \rightarrow)$ and a set of nodes $\hat{V} \subseteq V$, the controller-synthesis problem for G with respect to invariance (resp. reachability) of $\hat{V}$ is to find a strategy (if one exists) with respect to invariance (resp. reachability) of $\hat{V}$.

Remark 2. Notice that the controller-synthesis problems for reachability and invariance are *almost* dual. As we see below the algorithms for the two problems are similar, however, we cannot encode one problem as the negation of the other by simply changing controllable edges to uncontrollable and vice-versa. This is because of the assumption that the environment is "faster" than the controller: in a node where both controllable and uncontrollable actions are possible, our definition of strategy assumes that the environment can force an uncontrollable action before the controller can make a choice.

2.2 The On-the-Fly Algorithms for Invariance and Reachability

Based on the definition of strategies we derive two algorithms, shown in figures 3 and 4, for computing a strategy with respect to invariance and reachability, respectively. The algorithms are on-the-fly in the sense that they return a strategy as soon as one is found. Therefore, the input graph need not be fully generated/explored, which can result in significant savings in performance.

Both algorithms use a depth-first search (DFS) on the input GG. In the figures, the DFS is represented by procedures calling each other in a recursive manner. In practice, a stack is used to eliminate recursion and implement the DFS directly. This stack holds the currently visited path. The set of nodes which are in the stack is denoted *Stack*.

The DFS is adapted to the definition of a strategy, so that whenever a node is explored, all its uncontrollable successors and at least one controllable successor are also explored. Nodes are marked with a *controllability status* during the search. In the algorithm for invariance, a node is initially marked *maybe*, until it is found that it cannot be winning, whereupon its mark is updated to *no*. Dually, in the algorithm for reachability, a node is initially marked *maybe*, until it is found that it is winning, whereupon its mark is updated to *yes*. The sets *Maybe*, *No* and *Yes* are used to store visited nodes and represent their marks.

The algorithms also use a set of edges *Strat* representing the strategy. The set of edges *NegStrat* in the algorithm for invariance represents the "counter-strategy" showing how the environment can lead the system out of the set of safe nodes. This can be used as diagnostics in case a strategy for the controller does not exist. In the case of reachability, such a special structure is not needed, since the explored graph is also the counter-example.

The Algorithm for Invariance. Intuitively, the invariance algorithm works as follows. Procedure `Reach` explores the graph in depth-first order. For each newly visited node v, the uncontrollable successors of v are explored first. If not all of them are winning then v cannot be winning either, and control moves to procedure `UndoMaybe` (line 1). Otherwise, its controllable successors are explored by procedure `CheckControllable` (line 2). If none of them is winning then again v cannot be winning. Procedure `UndoMaybe` updates a node v which was falsely assumed to be winning, as well as all predecessors of v, since their computed strategies are no longer valid. In particular, all uncontrollable predecessors of v are not winning. Also, if w is a controllable predecessor of v, then a new controllable successor should be found for w (line 10). This is done by procedure `CheckControllable`, which explores the remaining controllable successors of v.

At the end of the algorithm, and if the answer is not *no*, then the sub-graph represented by the set of edges *Strat* contains the strategy. If the answer is *no*, then *NegStrat* contains a counter-example, that is, a "counter-strategy" showing that the controller has no way to avoid the environment leading the system to a bad state.

The Algorithm for Reachability. The algorithm for reachability works in a dual manner. A difference is that edges which are inserted in *Strat* are no longer removed. A node v is inserted in *Yes* (procedure `UndoMaybe`) only if all its uncontrollable successors and at least one controllable successor are already in *Yes*. When v is inserted in *Yes*, its predecessors are also updated: if v was

```
FindStrategyForInvariance $((V, v_0, \rightarrow), \hat{V})$ {
  $No := Maybe := \{\}$ ;
  $Strat := NegStrat := \{\}$ ;
  $Controllable := \{v_1 \xrightarrow{c} v_2 \mid v_1, v_2 \in V\}$ ;
  if (Reach($v_0$) $= no$) then return “No strategy exists” ;
  else return “Found strategy $Strat$” ;

  Reach($v$) {
    if ($v \in No$ or $v \notin \hat{V}$) then return $no$ ;
    if ($v \in Maybe$) then return $maybe$ ;
    $Maybe := Maybe \cup \{v\}$ ;   /* new node */
    for each ($v \xrightarrow{u} w$) do
      if (Reach($w$) $= no$) then
        $NegStrat := NegStrat \cup \{v \xrightarrow{u} w\}$ ;
        goto FAIL ;                                              (1)
    end for
    if (CheckControllable($v$) $\neq no$) then
      $Strat := Strat \cup \{v \xrightarrow{u} v' \mid v' \in V\}$ ;              (2)
      return $maybe$ ;
    else $NegStrat := NegStrat \cup \{v \xrightarrow{c} w \mid w \in V\}$ ;
  FAIL:
    UndoMaybe($v$) ;
    return $no$ ;
  }
  CheckControllable($v$) {
    while ($\exists v \xrightarrow{c} w \in Controllable$) do                       (3)
      $Controllable := Controllable \setminus \{v \xrightarrow{c} w\}$ ;
      if (Reach($w$) $\neq no$) then                                 (4)
        $Strat := Strat \cup \{v \xrightarrow{c} w\}$ ;                          (5)
        return $maybe$ ;
    end while
    return $no$ ;                                                    (6)
  }
  UndoMaybe($v$) {   /* Update from $Maybe$ to $No$ */
    $Maybe := Maybe \setminus \{v\}$ ;                                  (7)
    $No := No \cup \{v\}$ ;                                          (8)
    while ($\exists w \in Maybe \setminus Stack . w \rightarrow v \in Strat$) do          (9)
      $Strat := Strat \setminus \{w \rightarrow v\}$ ;
      $NegStrat := NegStrat \cup \{w \rightarrow v\}$ ;
      if ($w \xrightarrow{u} v$) then                                     (10)
        UndoMaybe($w$) ;
      elsif (CheckControllable($v$) $= no$) then
        UndoMaybe($w$) ;
    end while
  }
}
```

Fig. 3. On-the-fly controller synthesis for invariance.

the single "missing" successor for a node w to be winning (line 6), then procedure `UndoMaybe` is called recursively for w; otherwise, the remaining unexplored successors of w are explored (line 7). Another difference from the algorithm for invariance is that here the counter-example strategy is not explicitly shown: this is because the explored graph itself is a counter-example.

Example. Consider again the game graphs of figure 1. Suppose that the nodes are numbered in the order they are visited by a DFS (for instance, the edge $0 \xrightarrow{c} 1$ in the graph (a) is visited before the edge $0 \xrightarrow{c} 2$). Then, the on-the-fly algorithms presented above compute the strategies shown in figure 2 without exploring the whole graphs. In particular, the algorithm for invariance only visits nodes 0 and 1 of graph (a) and algorithm for reachability avoids visiting node 4 and the corresponding edges.

A more realistic example demonstrating the on-the-fly aspect of the algorithms is given in sections 3 and 4 where we apply the technique to controller synthesis for timed automata.

Complexity. The worst-case complexity of the algorithms is $O(n + m)$, where n and m are, respectively, the number of nodes and edges in the graph. Let us see why this is so, for the case of invariance. Each node and edge is considered at most twice: one time when they are inserted in *Maybe* or *Strat* and possibly a second time to be removed. This costs $O(n+m)$. In the worst case, when a node is removed by procedure `UndoMaybe`, all its previously explored predecessors which are not in the stack need to be examined (line 9 of figure 3). This also means that at most m predecessors are going to be considered during the backtracking procedure. In practice, the complexity of the algorithms can be reduced by using clever book-keeping to mark predecessors of a node that are likely to be updated.

3 Timed Automata and Time-Abstracting Bisimulations

In this section we briefly review the model of timed automata and define the time-abstracting bisimulation which reduces the infinite state space of timed automata into a finite graph which preserves most properties of interest. In the next section, we show how this graph can be used for controller synthesis in the timed context.

3.1 Timed Automata

Clocks, Bounds, and Polyhedra. Let R be the set of non-negative reals and $\mathcal{X} = \{x_1, ..., x_n\}$ be a set of variables in R, called *clocks*. An *$\mathcal{X}$-valuation* is a function $\mathbf{v} : \mathcal{X} \mapsto \mathsf{R}$. For some $X \subseteq \mathcal{X}$, $\mathbf{v}[X := 0]$ is the valuation $\mathbf{v}'$, such that $\forall x \in X \,.\, \mathbf{v}'(x) = 0$ and $\forall x \notin X \,.\, \mathbf{v}'(x) = \mathbf{v}(x)$. For every $\delta \in \mathsf{R}$, $\mathbf{v} + \delta$ is a valuation such that for all $x \in \mathcal{X}$, $(\mathbf{v} + \delta)(x) = \mathbf{v}(x) + \delta$.

```
FindStrategyForReachability $((V, v_0, \rightarrow), \hat{V})$ {
  $Yes$ := $Maybe$ := {} ;
  $Strat$ := $ExploredEdges$ := {} ;
  if (Reach($v_0$) = $yes$) then return “Found strategy $Strat$” ;
  else return “No strategy exists” ;

  Reach($v$) {
    if ($v \in Yes$ or $v \in \hat{V}$) then return $yes$ ;
    if ($v \in Maybe$) then return $maybe$ ;
    $Maybe := Maybe \cup \{v\}$ ;    /* new node */
    for each ($v \xrightarrow{u} w \notin ExploredEdges$) do
      $ExploredEdges := ExploredEdges \cup \{v \xrightarrow{u} w\}$ ;
      if (Reach($w$) $\neq$ $yes$) then return $maybe$ ;                    (1)
    end for
    if (CheckControllable($v$) = $yes$) then                          (2)
      UndoMaybe($v$) ;                                                (3)
      return $yes$ ;
    end if
    return $maybe$ ;
  }

  CheckControllable($v$) {
    for each ($v \xrightarrow{c} w \notin ExploredEdges$) do
      $ExploredEdges := ExploredEdges \cup \{v \xrightarrow{c} w\}$ ;
      if (Reach($w$) = $yes$) then
        $Strat := Strat \cup \{v \xrightarrow{c} w\}$ ;                          (4)
        return $yes$ ;
      end if
    end for
    return $maybe$ ;
  }

  UndoMaybe($v$) {    /* Update from $Maybe$ to $Yes$ */
    $Maybe := Maybe \setminus \{v\}$ ;
    $Yes := Yes \cup \{v\}$ ;
    $Strat := Strat \cup \{v \xrightarrow{u} v' \mid v' \in V\}$ ;                  (5)
    while ($\exists w \in Maybe \setminus Stack \,.\, w \rightarrow v$) do
      if ($w \xrightarrow{c} v \wedge \forall w \xrightarrow{u} v' \,.\, v' \in Yes$) then              (6)
        $Strat := Strat \cup \{w \xrightarrow{c} v\}$ ;
        UndoMaybe($w$) ;
      else                                                            (7)
        $Maybe := Maybe \setminus \{w\}$ ;
        if (Reach($w$) = $yes$) then UndoMaybe($w$) ;
      end if
    end while
  }
}
```

Fig. 4. On-the-fly controller synthesis for reachability.

A *bound* [Dil89] over $\mathcal{X}$ is a constraint of the form $x_i \sim c$ or $x_i - x_j \sim c$, where $1 \leq i \neq j \leq n$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathsf{N} \cup \{\infty\}$. An $\mathcal{X}$-valuation $\mathbf{v}$ satisfies the bound $x_i \sim c$ (resp. $x_i - x_j \sim c$) if $\mathbf{v}(x_i) \sim c$ (resp. $\mathbf{v}(x_i) - \mathbf{v}(x_j) \sim c$).

An *$\mathcal{X}$-polyhedron* ζ is a set of $\mathcal{X}$-valuations satisfying a conjunction of bounds over $\mathcal{X}$. We use the conjunction of bounds to refer to the $\mathcal{X}$-polyhedron itself, for instance, if ζ is the set of valuations satisfying $x \leq 5 \wedge x \leq y$ then we write $x \leq 5 \wedge x \leq y$ instead of $\{\mathbf{v} \mid \mathbf{v}(x) \leq 5 \wedge \mathbf{v}(x) \leq \mathbf{v}(y)\}$. If the bounds of ζ are unsatisfiable, ζ defines the empty valuation set.

Syntax of Timed Automata. A *timed automaton* [ACD93, HNSY94] (TA) is a tuple $A = (\mathcal{X}, Q, q_0, E, I)$, where:

- $\mathcal{X}$ is a finite set of clocks.
- Q is a finite set of *discrete states.*
- q_0 is the initial discrete state.
- E is a finite set of *edges* of the form $e = (q, \zeta, X, q')$, where $q, q' \in Q$ are the *source* and *target* discrete states, ζ is an $\mathcal{X}$-polyhedron, called the *guard* of e, and $X \subseteq \mathcal{X}$ is a set of clocks to be reset.
- I is a function associating at each discrete state $q \in Q$ an $\mathcal{X}$-polyhedron $I(q)$ called the *invariant* of q.

Given an edge $e = (q, \zeta, X, q')$,we write $\mathsf{guard}(e)$ and $\mathsf{reset}(e)$ for ζ and X, respectively. Given a discrete state q, we write $\mathsf{in}(q)$ for the set of edges of the form $(\_, \_, \_, q)$.

Semantics of Timed Automata. A *state* of A is a pair $(q, \mathbf{v})$, where $q \in Q$ is a location, and $\mathbf{v}$ is an $\mathcal{X}$-valuation such that $\mathbf{v} \in I(q)$. $s_0 = (q_0, \mathbf{0})$ is the initial state of A, where $\mathbf{0}$ is the valuation assigning zero to all clocks in $\mathcal{X}$.

The semantics of a TA A are given in terms of the *semantic graph* G_A, which is generally an infinite (non-enumerable) structure, due to the density of the time domain. The nodes of G_A are states of A, the initial node being s_0. G_A has two types of edges: *discrete* edges of the form $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')$, where $e \in E, e = (q, \zeta, X, q')$ such that $\mathbf{v} \in \zeta$ and $\mathbf{v}' = \mathbf{v}[X := 0]$; *time* edges of the form $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v} + \delta)$, where $\mathbf{v} + \delta \in I(q)$. This graph has by definition the following properties of *time continuity* and *additivity*:

$$s \xrightarrow{\delta} s + \delta \text{ implies } s \xrightarrow{\delta'} s + \delta', \text{ for all } \delta' < \delta \tag{1}$$

$$s \xrightarrow{\delta} s + \delta \text{ and } s + \delta \xrightarrow{\delta'} s + \delta + \delta' \text{ imply } s \xrightarrow{\delta + \delta'} s + \delta + \delta' \tag{2}$$

For simplicity, we consider in the sequel only *deadlock-free* TA, that is, where for each state s there exists some $\delta \in \mathsf{R}$ and an edge $e \in E$ such that $s \xrightarrow{\delta} s + \delta \xrightarrow{e} s'$.

Example. Timed automata can be composed in parallel, so that systems with more than one components can be described more easily. We do not define formally the parallel composition here, due to lack of space (see, for instance, [Tri98] for more details). Instead, we present a well-known example of a system composed by three TA (figure 5). The example is about a simple railway-crossing system where a controller commands a gate to lower and raise according to the arrivals and departures of a train. Assuming the usual parallel composition operation with synchronization of edges with same labels, the composite timed automaton modeling the global system is shown in figure 6.

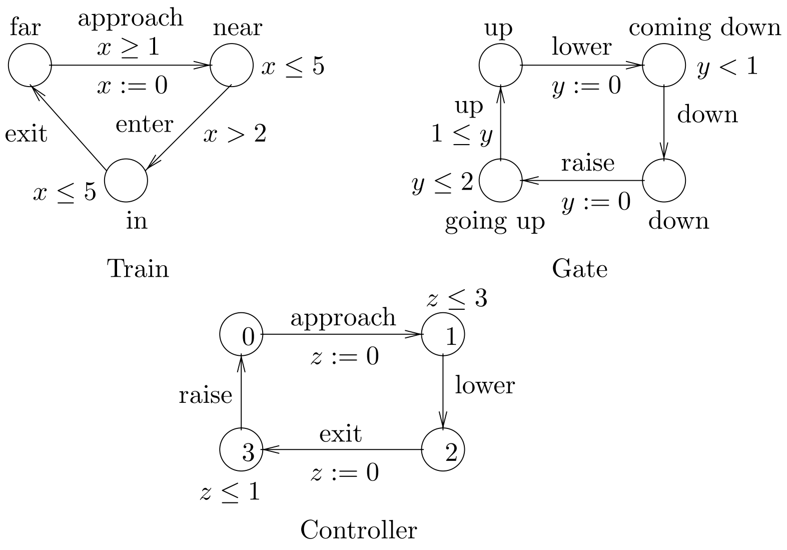

Fig. 5. The Train–Gate–Controller example.

3.2 Time-Abstracting Bisimulation and Quotient Graph

In order to apply algorithmic procedures to timed automata, we need a semantic model which is finite. For this purpose, we define the *time-abstracting bisimulation*, an equivalence which abstracts away from the quantitative aspect of time: we know that *some* time passes, but not how much. Given a TA, the time-abstracting bisimulation induces a finite graph, the *quotient*, which preserves all properties of interest, and can be therefore used for controller synthesis in the timed setting, as we show in the following section.

Time-Abstracting Bisimulation. Consider a TA A with set of edges E. A binary relation $\approx$ on the states of A is a (strong) *time-abstracting bisimulation* (TaB) if for all states $s_1 \approx s_2$, the following conditions hold:

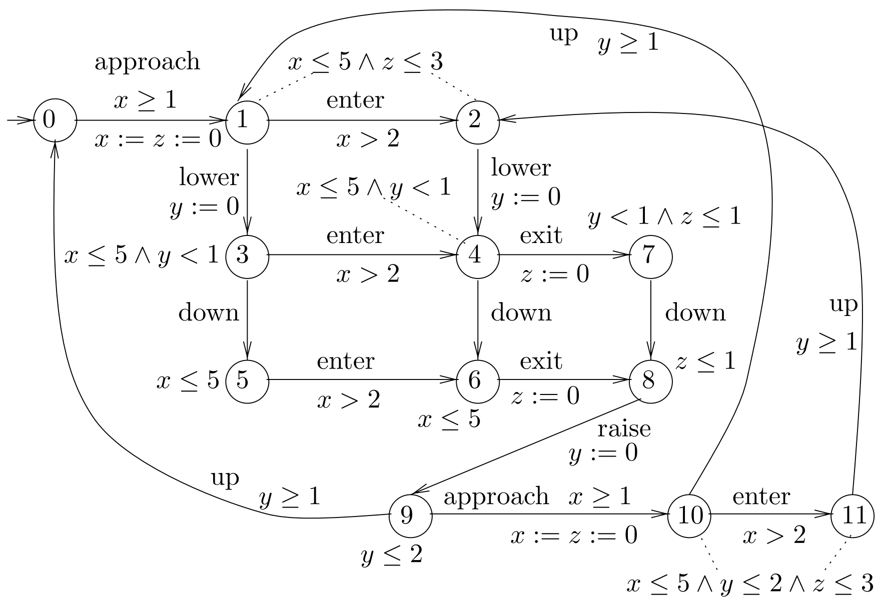

Fig. 6. The composite TA for the Train–Gate–Controller system.

1. if $s_1 \xrightarrow{e_1} s_3$, for some $e_1 \in E$, then there exists $e_2 \in E$ such that $s_2 \xrightarrow{e_2} s_4$ and $s_3 \approx s_4$;
2. if $s_1 \xrightarrow{\delta_1} s_3$ then there exists $\delta_2 \in \mathsf{R}$ such that $s_2 \xrightarrow{\delta_2} s_4$ and $s_3 \approx s_4$;
3. the above conditions also hold if the roles of s_1 and s_2 are reversed.

The definition is illustrated in figure 7 (left). The states s_1 and s_2 are said to be Ta-bisimilar. In general, two TA A_1 and A_2 are said to be Ta-bisimilar if there exists a TaB $\approx$ on the states of A_1 and A_2, such that $s_0^1 \approx s_0^2$, where s_0^i is the initial state of A_i.

Given a set of states $\hat{S}$ of A and a TaB $\approx$ on A, we say that $\approx$ *respects* $\hat{S}$ if for any $s_1 \approx s_2$, $s_1 \in \hat{S}$ iff $s_2 \in \hat{S}$.

$$\begin{tikzcd}
s_1 \arrow[r, dotted, no head] \arrow[d, "e_1"'] & s_2 \arrow[d, "e_2"] \\
s_3 \arrow[r, dotted, no head] & s_4
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
s_1 \arrow[r, dotted, no head] \arrow[d, "\delta_1"'] & s_2 \arrow[d, "\delta_2"] \\
s_3 \arrow[r, dotted, no head] & s_4
\end{tikzcd}$$

Fig. 7. Time-abstracting bisimulations.

Time-Abstracting Quotient Graph. Being an equivalence, a TaB induces a *partition* $\mathcal{C}$ of the state space of a TA into *equivalence classes*. We can prove that $\mathcal{C}$ has finite cardinality, by showing that the *region equivalence* of [ACD93] is a special case of time-abstracting bisimulation (in general, much stronger than necessary). Since the region equivalence induces a finite number of classes, so does the greatest time-abstracting bisimulation (for more details, see [Tri98]). We need this observation to conclude that the quotient graph of a TA (defined below) is finite.

Given a TA A and the greatest TaB $\approx$ on A, the *$\approx$-quotient* of A is the graph $G_A^{\approx} = (\mathcal{C}, C_0, \rightarrow_{\approx})$, such that:

- $\mathcal{C}$, the set of nodes of $G_A^{\approx}$ is the set of classes induced by $\approx$.
- C_0, the initial node of $G_A^{\approx}$ is the class containing s_0.
- $\rightarrow_{\approx}$ contains two types of edges corresponding to the discrete and time edges of the semantic graph of A. More precisely, for $C_1, C_2 \in \mathcal{C}$,
  1. if there exist $s_1 \in C_1, s_2 \in C_2, e \in E$ such that $s_1 \xrightarrow{e} s_2$, then $G_A^{\approx}$ has an edge $C_1 \xrightarrow{e}_{\approx} C_2$,
  2. if there exist $s \in C_1, \delta \in \mathsf{R}$ such that $s \xrightarrow{\delta} s + \delta$, $s + \delta \in C_2$ and $\forall \delta' < \delta \;.\; s + \delta' \in C_1 \cup C_2$, then $G_A^{\approx}$ has an edge $C_1 \xrightarrow{\tau} C_2$.
  3. if for all $s \in C_1, \delta \in \mathsf{R}$ such that $s \xrightarrow{\delta} s + \delta$, $s + \delta \in C_1$, then $G_A^{\approx}$ has an edge $C_1 \xrightarrow{\tau} C_1$.

  Notice that an edge $C_1 \xrightarrow{\tau} C_2$ of the quotient graph (item 2) represents the *continuous time passage* from states in C_1 to states in C_2. That is, from each state in C_1 time can pass until the system moves to C_2, without passing from any other class meanwhile. Also, for classes containing all their time successors we add a self-looping τ edge (item 3).

It is worth noting that other definitions of the quotient graph are possible, especially concerning the choices of the set of τ edges (for instance, we could consider taking the transitive closure of the τ-edge relation, which corresponds to the additivity of time successors in the semantic level). Defining the quotient graph as we did above is essential for the correctness of the method to reduce TA controller synthesis to controller synthesis for game graphs, presented in section 4.

A technique to generate the time-abstracting quotient of a TA has been presented in [TY96]. The technique consists in starting from an initial partition of the set of states (possibly respecting a set of initial constraints) and then *refining* the partition until it becomes *stable* with respect to discrete and time edges. The final stable partition induces an equivalence which is a TaB. The technique has been implemented in the module `minim`, part of the real-time verification tool KRONOS [DOTY96, BDM$^+$98].

Example. Applying `minim` to the Train–Gate–Controller system of figure 6, we obtain the quotient graph shown in figure 8 (self-looping τ-edges are not shown, for clarity).

Some of the nodes of the quotient graph are detailed in table 1. Being equivalence classes with an infinite number of states each, the nodes are shown in *symbolic* form $(\mathbf{q}, \zeta)$, where $\mathbf{q}$ is a vector of discrete states (one for each TA) and ζ is a polyhedron representing the set of valuations associated in this discrete state. In other words, $(\mathbf{q}, \zeta)$ represents the set of states $\{(\mathbf{q}, \mathbf{v}) \mid \mathbf{v} \in \zeta\}$.

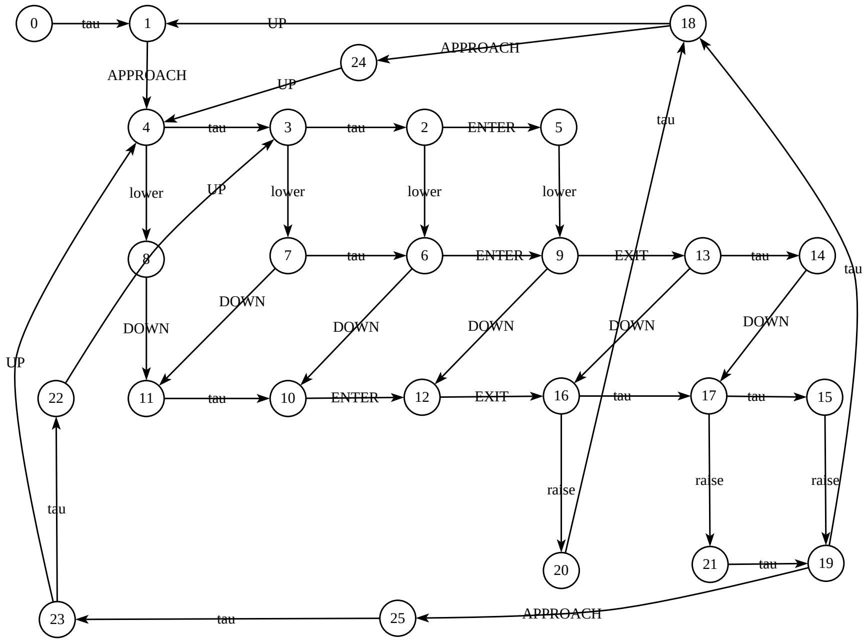

Fig. 8. The time-abstracting quotient graph of the Train–Gate–Controller example.

0:	(far,	up,	0, $x < 1$)
1:	(far,	up,	0, $x \geq 1$)
4:	(near,	up,	1, $x \leq 1 \wedge z < x + 1$)
8:	(near,	coming down,	2, $y < 1 \wedge x \leq y + 1 \wedge x < z + 2$)
10:	(near,	down,	2, $2 < x \leq 5$)
11:	(near,	down,	2, $x \leq 2$)
12:	(in,	down,	2, $x \leq 5$)
16:	(far,	down,	3, $x = 0 \wedge z \leq x$)
18:	(far,	going up,	0, $x \geq 1 \wedge 1 \leq y \leq 2$)
20:	(far,	going up,	0, $x < 1 \wedge x = y$)
24:	(near,	going up,	1, $y \leq 2 \wedge x + 1 \leq y \wedge z < x + 1$)

Table 1. Some of the nodes of the time-abstracting quotient of figure 8.

4 Timed Controller Synthesis

In this section we define the controller synthesis problem in the setting of timed automata and give a partially on-the-fly solution. The model is a natural extension of the untimed model of game graphs, namely, timed automata with controllable and uncontrollable edges. Our solution consists in generating the time-abstracting quotient graph of a TA, then applying to this graph (viewed as a game graph) the algorithms of section 2 to compute a timed strategy. The method is only half on-the-fly since it relies on the generation of the quotient first, before the on-the-fly search for a strategy is applied.

4.1 The Model: Game Timed Automata

A *game timed automaton* [MPS95] (GTA) is a TA whose set of edges E is partitioned into two disjoint sets E^c (the controllable edges) and E^u (the uncontrollable edges).

Example. Consider again the Train-Gate-Controller system of figure 5. Regarding the Train and Gate automata as being the environment, we can view the composite TA of figure 6 as a GTA where edges labeled "lower" or "raise" are controllable, while the rest are uncontrollable.

Timed Strategies. The semantics of a GTA A are given in terms of *timed strategies*, which are extensions of strategies to account for the density of the time domain. Consider a GTA A, its semantic graph G_A, and a subset of its states $\hat{S}$.

A timed strategy with respect to invariance of $\hat{S}$ is a sub-graph G_1 of G_A, which satisfies the time continuity and additivity conditions 1 and 2, as well as the following conditions:

1. s_0 (the initial state of A) is the initial node of G_1.
2. If s is a node of G_1 and $s \xrightarrow{e_u} s'$ is an edge of G_A, for some $e_u \in E^u$, then $s \xrightarrow{e_u}_1 s'$ is an edge of G_1.
3. If s is a node of G_1 and $s \xrightarrow{\delta} s + \delta$ is an edge of G_A, for some $\delta \in \mathsf{R}$, then:
   - either $s \xrightarrow{\delta}_1 s + \delta$ is an edge of G_1,
   - or there exist $\delta' < \delta$ and $e_c \in E^c$, such that $s \xrightarrow{\delta'}_1 s + \delta'$ and $s + \delta' \xrightarrow{e_c}_1 s''$ are edges of G_1.
4. Every node of G_1 belongs to $\hat{S}$.

(Notice that when we say that $s \rightarrow_1 s'$ is a node of G_1, this implies that s' must be an edge of G_1.) Intuitively, condition 2 makes sure that the controller does not "cheat" : if the environment can make a move in the original graph then it can also make this move in the strategy graph. Condition 3 deals with the passage of time : if δ time units can elapse in the original graph then δ time units should be able to elapse in the strategy graph, unless if the controller can make a move earlier, at $\delta' < \delta$. Finally, condition 4 ensures that the system remains in the set of safe states.

A timed strategy with respect to reachability of $\hat{S}$ is a sub-graph G_1 of G_A, which satisfies the time continuity and additivity conditions 1 and 2, as well as the following conditions:

1. s_0 (the initial state of A) is the initial node of G_1.
2. If $s \notin \hat{S}$ is a node of G_1 and $s \xrightarrow{e_u} s'$ is an edge of G_A, for some $e_u \in E^u$, then $s \xrightarrow{e_u}_1 s'$ is an edge of G_1.
3. If $s \notin \hat{S}$ is a node of G_1 and $s \xrightarrow{\delta} s + \delta$ is an edge of G_A, for some $\delta \in \mathsf{R}$, then:
   - either $s \xrightarrow{\delta}_1 s + \delta$ is an edge of G_1,
   - or there exist $\delta' < \delta$ and $e_c \in E^c$, such that $s \xrightarrow{\delta'}_1 s + \delta'$ and $s + \delta' \xrightarrow{e_c}_1 s''$ are edges of G_1.
4. For each node s of G_1 there exists a path $s \xrightarrow{\delta_1}_1 \xrightarrow{e_1}_1 s_1 \xrightarrow{\delta_2}_1 \xrightarrow{e_2}_1 \cdots \xrightarrow{\delta_n}_1 \xrightarrow{e_n}_1 s_n$ such that $e_1, ..., e_n \in E^c$ and $s_n \in \hat{S}$.

Conditions 2 and 3 differ from the case of invariance strategies in that for the target states $\hat{S}$ there is no requirement to continue the game. Condition 4 ensures that the controller can lead the system to the set of target states.

Controller Synthesis for Game Timed Automata. Given a GTA A and a set of states $\hat{S}$, the controller-synthesis problem for A with respect to invariance (resp. reachability) to find a timed strategy (if one exists) with respect to invariance (resp. reachability) of $\hat{S}$.

4.2 Reducing Game-Timed-Automata Synthesis to On-the-Fly Game-Graph Synthesis

We are now going to use the on-the-fly algorithms of section 2 to solve the controller-synthesis problem for GTA. Consider a GTA A with set of edges $E = E^c \cup E^u$ and a set $\hat{S}$ of states of A. Let $G_A^{\approx} = (\mathcal{C}, C_0, \rightarrow_{\approx})$ be the quotient graph of A with respect to the greatest time-abstracting bisimulation respecting $\hat{S}$.

From $G_{\approx}$ we build the game graph $G = (\mathcal{C}, C_0, \xrightarrow{c} \cup \xrightarrow{u})$, where $\xrightarrow{c}$ and $\xrightarrow{u}$ are constructed as follows:

1. For each edge $C \xrightarrow{e}_{\approx} C'$, for some $e \in E^c$, the edge $C \xrightarrow{c} C'$ is added to G.
2. For each edge $C \xrightarrow{e}_{\approx} C'$, for some $e \in E^u$, the edge $C \xrightarrow{u} C'$ is added to G.
3. For each edge $C \xrightarrow{\tau}_{\approx} C'$, the edge $C \xrightarrow{c} C'$ is added to G.

In other words, discrete transitions do not change controllability status, while time transitions are considered controllable. The intuition behind this choice is the following. Consider a τ-edge $C \xrightarrow{\tau}_{\approx} C'$. There are two possibilities:

– Either C also has a controllable discrete edge $C \xrightarrow{e_c}_{\approx} C''$, $e_c \in E^c$. Then the controller has a choice, either to let time pass, waiting for the environment to make a move (this comes down to picking $C \xrightarrow{\tau}_{\approx} C'$ as the controllable edge), or moving to C'' (this comes down to picking $C \xrightarrow{e_c}_{\approx} C''$). Thus, $C \xrightarrow{\tau}_{\approx} C'$ can be considered controllable.
– Or C has no controllable discrete edge in the quotient graph. Then the controller has no choice but to wait for the environment to make a move (recall that the TA is assumed deadlock-free). Therefore, also in this case, $C \xrightarrow{\tau}_{\approx} C'$ can be considered controllable as it is the controller's only choice.

We claim that the above construction is enough to reduce timed controller synthesis to the untimed case. Let $\hat{\mathcal{C}} = \{C \mid C \subseteq \hat{S}\}$ (i.e., $\hat{\mathcal{C}}$ is the set of all classes whose states satisfy $\hat{S}$).

Proposition 1. *A has a strategy with respect to invariance (resp. reachability) of $\hat{S}$ iff G has a strategy with respect to invariance (resp. reachability) of $\hat{\mathcal{C}}$.*

The strategy of G corresponds to a timed strategy of A given in symbolic form. At each symbolic state (class) the controller chooses either to let time pass (τ-edge) or make a move (e-edge). In the latter case, the move can also be delayed (that is, the strategy is not time-deterministic) as long as the system remains in the same symbolic state.

Example. We illustrate the on-the-fly algorithm for invariance on the Train–Gate–Controller example of figure 6. The game graph corresponding to this system is obtained from the quotient graph of figure 8 by simply marking all edges labeled "tau", "lower" or "raise" as controllable and the rest as uncontrollable.

We are interested in computing a controller keeping the system in a set of safe states where whenever the train is in the crossing the gate is down. That is, we want to solve the controller synthesis problem with respect to the above invariance property. The property holds at all nodes of the graph of figure 8 except nodes 5 and 9.

Based on proposition 1, we solve the problem by applying the algorithm of figure 3 on the game graph of figure 8, with set of safe nodes $\hat{V} = \overline{\{5, 9\}}$.

Assuming that in the DFS order (procedure `CheckControllable`) the "tau"-edges are explored last [1], the algorithm yields the strategy shown in figure 9. In node 4, the controller chooses the action "lower" instead of the "tau"-edge and a similar choice is made in node 16. For each node, all its uncontrollable successors are included in the strategy, according to the semantics (for instance, see node 18). It is worth noticing that this strategy corresponds exactly to the part of the graph explored during the DFS, which demonstrates that the algorithm is on-the-fly: no other nodes except those belonging to the computed strategy were needed to be explored, thus no other nodes were visited.

[1] This can sometimes be a good heuristic, since it corresponds to exploring first the "as-soon-as-possible" policy: indeed, "tau"-edges correspond to the passage of time (while the controller is waiting).

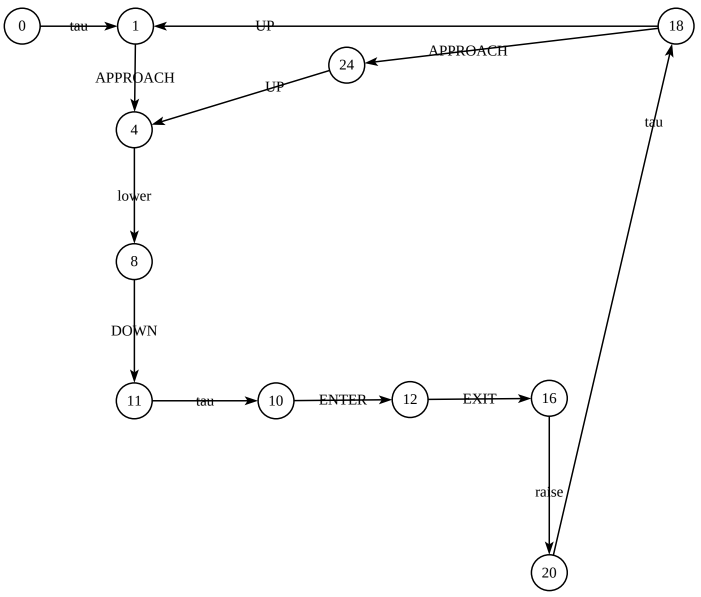

Fig. 9. A strategy for the graph of figure 8.

Examining the symbolic states corresponding to the nodes of the strategy (see table 1) we get some intuition about the timing constraints induced by the controller. For instance, node 4 corresponds to the symbolic state (near, up, 1, $x \leq 1 \wedge z < x + 1$). The discrete part (near, up, 1) corresponds to the Controller having just received the signal "approach" from the Train. The bound $z < x + 1$ means that the Controller waits less than 1 time unit before sending the command "lower" to the gate (whereas in the initial automaton of figure 5, it could wait up to 3 time units).

In a more methodological way, we can use the symbolic representation of the timed strategy in order to *restrict* the input GTA and obtain a *closed* system (but still, possibly non-deterministic) which satisfies the given invariance/reachability property. Restricting the initial GTA means replacing, for each controllable edge $(q, \zeta, X, q') \in E^c$, its guard ζ by $\zeta \cap \zeta'$, where (q, ζ') is the node in the strategy corresponding to the discrete state q. (If there is no winning node corresponding to q, then ζ can be replaced by the empty polyhedron.) The approach is explained in more detail in [Alt98].

5 Conclusions

We have presented on-the-fly techniques for controller synthesis of untimed and timed systems with respect to invariance and reachability. The technique in the untimed case uses DFS-based algorithms which return a strategy as soon as one is computed. In the timed case the technique relies on the generation of the time-abstracting quotient of a timed automaton. The quotient can be viewed as an untimed graph, and the previous algorithms can be applied on it to solve the timed controller-synthesis problem. Although the worst-case complexity of the algorithms is quadratic in the size of the graph (i.e., same as in [RW87]), their on-the-fly nature (illustrated in some toy examples) proves their practical interest.

References

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[Alt98] K. Altisen. Génération automatique d'ordonnancements pour systèmes temporisés. Technical report, Mémoire de DEA, Ensimag, Grenoble, 1998. In french.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Department of Computer Science, Stanford University, 1991.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, 1995.

[BDM$^+$98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In *CAV'98*, 1998.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[HW91] G. Hoffmann and H. Wong Toi. The input-output control of real-time discrete event systems. In *30th IEEE Conf. on Decision and Control*, 1991.

[HW92] G. Hoffmann and H. Wong Toi. Symbolic synthesis of supervisory controllers. In *American Control Conference*, 1992.

[Le 93] M. Le Borgne. *Dynamical Systems over finite fields.* PhD thesis, Université de Rennes, 1993. In French.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS '95*, 1995.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symp. POPL*, 1989.

[RW87] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), January 1987.

[Tri98] S. Tripakis. *The formal analysis of timed systems in practice.* PhD thesis, Université Joseph Fourrier de Grenoble, 1998.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time–abstracting bisimulations. In *Proc. 8th Conference Computer-Aided Verification, CAV'96, Rutgers, NJ*, volume 1102 of *LNCS*, pages 232–243. Springer-Verlag, July 1996.

[WM99] W. Wonham and C. Meder. The TTCT tool. Personal communication, 1999.